**Subject:**

**High Performance Computing**

**Subject Code:**

**410250**

**Group A**

**Experiment No.: 1**

**Title of the Assignment:** Design and implement Parallel Breadth First Search and Depth First Search

based on existing algorithms using OpenMP. Use a Tree or an undirected graph for BFS and DFS

**Objective of the Assignment:** Students should be able to Write a program to implement Parallel Breadth First Search and Depth First Search based on existing algorithms using OpenMP

**Prerequisite:**

1. Basic of programming language

2. Concept of BFS and DFS

3. Concept of Parallelism

**Theory:**

1. Concept of BFS And DFS

3. Concept of OpenMP

4. Code Explanation with Output

**What is BFS?**

BFS stands for Breadth-First Search. It is a tree/graph traversal algorithm used to explore all the nodes of a tree/graph systematically, starting from the root node or a specified starting point, and visiting all the neighboring nodes at the current depth level before moving on to the next depth level. The algorithm uses a queue data structure to keep track of the nodes that need to be visited, and marks each visited node to avoid processing it again.

**What is DFS?**

DFS stands for Depth-First Search. It is a popular tree/graph traversal algorithm that explores as far as possible along each branch before backtracking. This algorithm can be used to find the shortest path between two vertices or to traverse a graph in a systematic way.

**Concept of OpenMP**

● OpenMP (Open Multi-Processing) is an application programming interface (API) that supports shared-memory parallel programming in C, C++, and Fortran. It is used to write parallel programs that can run on multi-core processors, multiprocessor systems, and parallel computing clusters.

● OpenMP provides a set of directives and functions that can be inserted into the source code of a program to parallelize its execution. These directives are simple and easy to use, and they can be applied to loops, sections, functions, and other program constructs. The compiler then generates parallel code that can run on multiple processors concurrently.

● OpenMP uses a fork-join model of parallel execution, where a master thread forks multiple worker threads to execute a parallel region of the code, and then waits for all threads to complete before continuing with the sequential part of the code.

● OpenMP is widely used in scientific computing, engineering, and other fields that require high-performance computing. It is supported by most modern compilers and is available on a wide range of platforms, including desktops, servers, and supercomputers.

**How Parallel BFS Work**

● Parallel BFS (Breadth-First Search) is an algorithm used to explore all the nodes of a graph or tree systematically in parallel.

● The parallel BFS algorithm starts by selecting a root node or a specified starting point, and then assigning it to a thread or processor in the system. Each thread maintains a local queue of nodes to be visited and marks each visited node to avoid processing it again.

● The algorithm then proceeds in levels, where each level represents a set of nodes that are at a certain distance from the root node. Each thread processes the nodes in its local queue at the current level, and then exchanges the nodes that are adjacent to the current level with other threads or processors. This is done to ensure that the nodes at the next level are visited by the next iteration of the algorithm.

● **The parallel BFS algorithm uses two phases:**

The computation phase and the communication phase.

In the computation phase, each thread processes the nodes in its local queue, while in the communication phase, the threads exchange the nodes that are adjacent to the current level with other threads or processors.

● The parallel BFS algorithm terminates when all nodes have been visited or when a specified node has been found. The result of the algorithm is the set of visited nodes or the shortest path from the root node to the target node.

● Parallel BFS can be implemented using different parallel programming models, such as OpenMP, MPI, CUDA, and others. The performance of the algorithm depends on the number of threads or processors used, the size of the graph, and the communication overhead between the threads or processors.

**Conclusion:**

In this way we can achieve parallelism while implementing Breadth First Search and Depth First Search

**Experiment No.: 2**

**Title of the Assignment:** Write a program to implement Parallel Bubble Sort. Use existing algorithms and measure the performance of sequential and parallel algorithms.

**Objective of the Assignment:** Students should be able to Write a program to implement Parallel Bubble Sort and can measure the performance of sequential and parallel algorithms.

**Prerequisite:**

4. Basic of programming language

5. Concept of Bubble Sort

6. Concept of Parallelism

**Contents for Theory:**

1. Concept and Use of Bubble Sort

4. How Parallel Bubble Sort Work

5. How to measure the performance of sequential and parallel algorithms?

**What is Bubble Sort?**

Bubble Sort is a simple sorting algorithm that works by repeatedly swapping adjacent elements if they are in the wrong order. It is called "bubble" sort because the algorithm moves the larger elements towards the end of the array in a manner that resembles the rising of bubbles in a liquid.

The basic algorithm of Bubble Sort is as follows:

1. Start at the beginning of the array.

2. Compare the first two elements. If the first element is greater than the second element, swap them.

3. Move to the next pair of elements and repeat step 2.

4. Continue the process until the end of the array is reached.

5. If any swaps were made in step 2-4, repeat the process from step 1.

The time complexity of Bubble Sort is O(n^2), which makes it inefficient for large lists. However, it has the advantage of being easy to understand and implement, and it is useful for educational purposes and for sorting small datasets.

.

**How Parallel Bubble Sort Work**

● Parallel Bubble Sort is a modification of the classic Bubble Sort algorithm that takes advantage of parallel processing to speed up the sorting process.

● In parallel Bubble Sort, the list of elements is divided into multiple sub-lists that are sorted concurrently by multiple threads. Each thread sorts its sub-list using the regular Bubble Sort algorithm. When all sub-lists have been sorted, they are merged together to form the final sorted list.

● The parallelization of the algorithm is achieved using OpenMP, a programming API that supports parallel processing in C++, Fortran, and other programming languages.

● In the parallel Bubble Sort algorithm, the main loop that iterates over the list of elements is divided into multiple iterations that are executed concurrently by multiple threads. Each thread sorts a subset of the list, and the threads synchronize their work at the end of each iteration to ensure that the elements are properly ordered.

**How to measure the performance of sequential and parallel algorithms?**

To measure the performance of sequential Bubble sort and parallel Bubble sort algorithms,

you can follow these steps:

1. Implement both the sequential and parallel Bubble sort algorithms.

2. Choose a range of test cases, to test the performance of both algorithms.

3. Use a reliable timer to measure the execution time of each algorithm on each test case.

4. Record the execution times and analyze the results.

When measuring the performance of the parallel Bubble sort algorithm, you will need to specify the number of threads to use. You can experiment with different numbers of threads to find the optimal value for your system.

**Conclusion**-

In this way we can implement Bubble Sort in parallel way using OpenMP also come to know how to measure performance of serial and parallel algorithm.

**Experiment No: 3**

**Title of the Assignment**: Implement Min, Max, Sum and Average operations using Parallel Reduction.

**Objective of the Assignment**: To understand the concept of parallel reduction and how it can be used to perform basic mathematical operations on given data sets.

**Prerequisite:**

1. Parallel computing architectures

2. Parallel programming models

3. Proficiency in programming languages

4. Familiarity with a parallel programming library or framework, such as OpenMP, MPI, or CUDA.

5. A suitable parallel programming environment, such as a multi-core CPU, a cluster of computers, or a GPU.

**Theory:**

1. What is parallel reduction and its usefulness for mathematical operations on large data?

2. Concept of OpenMP , CUDA

3. How do parallel reduction algorithms for Min, Max, Sum, and Average work, and what are their advantages and limitations?

**Parallel Reduction**.

The binary tree reduction algorithm can be implemented using various parallel programming models, such as OpenMP, MPI, or CUDA. In OpenMP, the algorithm can be implemented using the parallel and for directives for parallelizing the computation, and the reduction clause for combining the partial results.

Here's a function-wise manual on how to understand and run the sample C++ program that demonstrates how to implement Min, Max, Sum, and Average operations using parallel reduction.

**1. Min\_Reduction function**

• The function takes in a vector of integers as input and finds the minimum value in the vector using parallel reduction. • The OpenMP reduction clause is used with the "min" operator to find the minimum value across all threads.

• The minimum value found by each thread is reduced to the overall minimum value of the entire array.

• The final minimum value is printed to the console.

**2. Max\_Reduction function**

• The function takes in a vector of integers as input and finds the maximum value in the vector using parallel reduction.

• The OpenMP reduction clause is used with the "max" operator to find the maximum value across all threads.

• The maximum value found by each thread is reduced to the overall maximum value of the entire array.

• The final maximum value is printed to the console.

**3. Sum\_Reduction function**

• The function takes in a vector of integers as input and finds the sum of all the values in the vector using parallel reduction.

• The OpenMP reduction clause is used with the "+" operator to find the sum across all threads.

• The sum found by each thread is reduced to the overall sum of the entire array.

• The final sum is printed to the console.

**4. Average\_Reduction function**

• The function takes in a vector of integers as input and finds the average of all the values in the vector using parallel reduction.

• The OpenMP reduction clause is used with the "+" operator to find the sum across all threads

• The sum found by each thread is reduced to the overall sum of the entire array.

• The final sum is divided by the size of the array to find the average.

• The final average value is printed to the console.

**5. Main Function**

• The function initializes a vector of integers with some values.

• The function calls the min\_reduction, max\_reduction, sum\_reduction, and average\_reduction functions on the input vector to find the corresponding values.

• The final minimum, maximum, sum, and average values are printed to the console. 6. Compiling and running the program Compile the program: You need to use a C++ compiler that supports OpenMP, such as g++ or clang. Open a terminal and navigate to the directory where your program is saved. Then, compile the program using the following command:

**$ g++ -fopenmp program.cpp -o program**

This command compiles your program and creates an executable file named "program".

The "- fopenmp" flag tells the compiler to enable OpenMP.

Run the program:

To run the program, simply type the name of the executable file in the terminal and press Enter:

**$ ./program**

**Conclusion:**

We have implemented the Min, Max, Sum, and Average operations using parallel reduction in C++ with OpenMP.

**Experiment 4(A)**

**Title of the Assignment**: Write a CUDA Program for Addition of two large vectors

**Objective of the Assignment:** Students should be able to perform CUDA Program for Addition of two large vectors

**Prerequisite:**

1. CUDA Concept

2. Vector Addition

3. How to execute Program in CUDA Environment

**Theory:**

1. Concept of CUDA

2. Addition of two large Vectors

3. Execution of CUDA Environment

**What is CUDA**

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA. It allows developers to use the power of NVIDIA graphics processing units (GPUs) to accelerate computation tasks in various applications, including scientific computing, machine learning, and computer vision.

CUDA provides a set of programming APIs, libraries, and tools that enable developers to write and execute parallel code on NVIDIA GPUs. It supports popular programming languages like C, C++, and Python, and provides a simple programming model that abstracts away much of the low-level details of GPU architecture. Using CUDA, developers can exploit the massive parallelism and high computational power of GPUs to accelerate computationally intensive tasks, such as matrix operations, image processing, and deep learning. CUDA has become an important tool for scientific research and is widely used in fields like physics, chemistry, biology, and engineering.

**Steps for Addition of two large vectors using CUDA**

1. Define the size of the vectors: In this step, you need to define the size of the vectors that you want to add. This will determine the number of threads and blocks you will need to use to parallelize the addition operation.

2. Allocate memory on the host: In this step, you need to allocate memory on the host for the two vectors that you want to add and for the result vector. You can use the C malloc function to allocate memory.

3. Initialize the vectors: In this step, you need to initialize the two vectors that you want to add on the host. You can use a loop to fill the vectors with data.

4. Allocate memory on the device: In this step, you need to allocate memory on the device for the two vectors that you want to add and for the result vector. You can use the CUDA function cudaMalloc to allocate memory.

5. Copy the input vectors from host to device: In this step, you need to copy the two input vectors from the host to the device memory. You can use the CUDA function cudaMemcpy to copy the vectors.

6. Launch the kernel: In this step, you need to launch the CUDA kernel that will perform the addition operation. The kernel will be executed by multiple threads in parallel. You can use the <<>> syntax to specify the number of blocks and threads to use.   
7. Copy the result vector from device to host: In this step, you need to copy the result vector from the device memory to the host memory. You can use the CUDA function cudaMemcpy to copy the result vector.

8. Free memory on the device: In this step, you need to free the memory that was allocated on the device. You can use the CUDA function cudaFree to free the memory.

9. Free memory on the host: In this step, you need to free the memory that was allocated on the host. You can use the C free function to free the memory.

**Execution of Program over CUDA Environment:**

Here are the steps to run a CUDA program for adding two large vectors:

1. Install CUDA Toolkit: First, you need to install the CUDA Toolkit on your system. You can download the CUDA Toolkit from the NVIDIA website and follow the installation instructions provided.

2. Set up CUDA environment: Once the CUDA Toolkit is installed, you need to set up the CUDA environment on your system. This involves setting the PATH and LD\_LIBRARY\_PATH environment variables to the appropriate directories.

3. Write the CUDA program: You need to write a CUDA program that performs the addition of two large vectors. You can use a text editor to write the program and save it with a .cu extension.

4. Compile the CUDA program: You need to compile the CUDA program using the nvcc compiler that comes with the CUDA Toolkit. The command to compile the program is:

**nvcc –o program\_name program\_name.cu**

5. This will generate an executable program named program\_name. Run the CUDA program:

Finally, you can run the CUDA program by executing the executable file generated in the previous step. The command to run the program is:

**./program\_name**

This will execute the program and perform the addition of two large vectors.

**Conclusion:**

Hence we have implemented Addition of two large vectors and Matrix Multiplication using CUDA.